E0 250 - Deep Learning

Project 3 - SNLI Dataset Classification

Deepesh Virendra Hada
M.Tech, Department of CSA
SR: 17196

15 May, 2020

1 Data

In this project, we use the *Stanford Natural Language Inference* (SNLI) corpus as our data. The SNLI corpus is a collection of 570k human-written English sentence pairs manually labeled for balanced classification with the labels *entailment*, *contradiction* and *neutral*, supporting the task of natural language inference. The dataset is split into three: *training*, *validation* and *test* sets, available as both *txt* and *JSON* files. Here are the number of examples:

- Training pairs: 550152
- Dev pairs: 10000
- Test pairs: 10000
- **Total pairs**: 570152

A pattern in this dataset consists of many fields, of which three are of importance to us for the task:

- **sentence1**: This sentence is the premise.
- **sentence2**: This sentence is the hypothesis. The sentiment of this hypothesis could either entail the premise, contradict it or just be neutral.
- **gold_label**: This is the label, or the class, which is used for training the model. This field must not be used during the testing phase.

2 Task

We implement a logistic regression model and a deep model for text classification on the *Stanford Natural Language Inference* (SNLI) corpus. The task is to classify an input pair of sentences (premise and hypothesis) into 3 possible classes, namely *entailment*, *contradiction* and *neutral*. In this project, we perform this task using the following two approaches:

- Logistic Regression using TF-IDF features using the *scikit-learn* library.
- LSTM-based Recurrent Neural Network Model using a deep learning library. We'll be using *PyTorch* for this.

We now see the two approaches in the following sections.

3 Logistic Regression using TF-IDF features

In this section, we discuss using a *Logistic Regression* based model using *TF-IDF* features to perform text classification on the SNLI corpus. To read the sentence pairs from the corpus, we have used the *jsonline* library to read the corpus from the JSON files, line-by-line. Each line in these JSON files consists of the training/test pattern. The overall vocabulary was constructed through the *training* and *validation* sets. The model was trained only on the *training* set, and was tested on the *test* set. We now see how the model was trained.

3.1 Preprocessing

A training example, which is retrieved from the JSON file of the training set, consists of many fields, of which three are of importance to us for the task: We retrieve a training example from the JSON file of the training set. The example gives us 3 important fields: the premise, hypothesis and the associated label. Before constructing the model, we perform some **preprocessing** so as to make the inputs agreeable to the model and remove *noise* from the sentences.

We first concat the premise and the hypothesis, so that the combined sentence represents a single input pattern. We now create a matrix X, which contains the entire training set. While being initially empty, each of the concatenated sentences are appended to this matrix. Similarly, we also form a vector Y, which contains the *gold_labels* of the corresponding entries in X.

We repeat this procedure for the validation(dev) and the test sets as well.

Next, we try to diminish the effect of noise. Noise is inherent in any language sentence and it appears in the form of punctuations, lower/upper cases, stop words, etc. These noisy elements add little meaning to the sentence and are just important to maintain the structure of the sentence. Hence, removing them is preferable. The following preprocessing was done on each of the training and test inputs (note that an input is a concatenation of premise and hypothesis):

- **Punctuations:** Punctuations like "comma", "apostrophe", "colon", etc. add structure to a sentence, but don't add any meaning to it, unlike entities like nouns and verbs. Hence, removing them is a desirable step.
- **Letter case:** Mixed case letters could be problematic as an uppercase letter and its lowercase counterpart are treated differently by modern programming languages. A simple preprocessing to maintain uniformity is to convert all the letters in all the inputs to lowercase letters.
- **Stop words:** Stop words are a set of commonly used words. For example, in English, "the", "is" and "and", would easily qualify as stop words. These words do not help in our task of text classification at all, and just like punctuations, help in maintaining the structure of the sentence. The set of stopwords was obtained from both the NLTK library and SKLearn's ENGLISH_STOP_WORDS and these words were then removed from all the inputs.

3.2 Tokenization

Given a character sequence and a defined document unit, *tokenization* is the task of chopping it up into pieces, called *tokens*, perhaps throwing away certain characters at the same time, such as punctuations, that we had discussed earlier. Now, it may happen that during test time, we see a new word, which does not form a part of our vocabulary, i.e., we didn't see that in the training time. In such cases, we replace that word with a special token, called the *¡UNK¿* token. In our model, we've used the *word_tokenize* method of the *NLTK* library.

3.3 Text Normalization

Stemming and *Lemmatization* are Text/Word Normalization techniques that are used to prepare text, words, and documents for further processing.

Stemming is the process of reducing inflection in words to their root forms, such as mapping a group of words to the same stem even if the stem itself is not a valid word in the language. So,

stemming words in a sentence may result in words that are not actual English words, and this poses a drawback.

Lemmatization, unlike Stemming, reduces the inflected words properly ensuring that the root word belongs to the language. In Lemmatization, the root word is called a *lemma*. A lemma (plural lemmas or lemmata) is the canonical form, dictionary form, or citation form of a set of words. For example, *runs*, *running*, *ran* are all forms of the word *run*, therefore *run* is the lemma of all these words. Because lemmatization returns an actual word of the language, we've used it for text normalization instead of Stemming.

3.4 TF-IDF Features

TF-IDF, short for *term frequency - inverse document frequency*, is a numerical statistic that reflects how important a word is to a document in a corpus. Using TF-IDF, we can score the relative importance of words in the entire corpus. It consists of two parts: term frequency; and inverse document frequency.

1. **Term Frequency:** This is given as the number of times a word appears in a document divded by the total number of words (not unique, **total**) in the document. Also, every document has its own term frequency. $tf_{i,j}$ is given as follows:

$$tf_{i,j} = \frac{n_{i,j}}{\sum n_{i,j}} \tag{1}$$

2. **Inverse Document Frequency:** Inverse document frequency determines the weight of rare words across all documents in the corpus. IDF is given as the log of the number of documents, N divided by the number of documents that contain the word w.

$$idf(w) = log(\frac{N}{df_t}) \tag{2}$$

Lastly, the TF-IDF is simply the TF multiplied by IDF.

$$tfidf_{i,j} = tf_{i,j} \times log\left(\frac{N}{df_t}\right) \tag{3}$$

3.5 Model

To perform the fit, a simple *Logistic Regression* is performed.

Recall that Logistic Regression is generally used when the target variable is categorical, unlike real-valued targets in the case of Linear Regression.

Since there are 3 possible classes in our task, we've used Multinomial Logistic Regression.

SKLearn provides us the tools to perform Logistic Regression. In the multiclass case like ours, the training algorithm uses the one-vs-rest (OvR) scheme, and uses the cross-entropy loss. Also, regularization is applied by default and it can handle both dense and sparse inputs.

We run the model till it converges, with the maximum number of iterations being set to $100,000$. The convergence however takes place much earlier than that.

3.6 Experiments

The model was tested with a lot many hyperparameter tunings, including various preprocessing strategies. However, no matter what the strategy was, the model seemed to saturate with a training accuracy close to 60 % and test accuracy of around 56 %. The following text normalization strategies were examined:

- Lemmatization vs Stemming: Lemmatization performed marginally better than stemming, resulting in an accuracy of 56.53 % as against 56.34 % on the test set, with the rest of the hyperparameter values being the same.
- Number of iterations: The number of iterations didn't matter in our case, as the model converged much before the max number of iterations set. The overall CPU time taken by the system for convergence was close to 4 minutes, 4 seconds.

3.7 Performance Measure

The following accuracies were obtained by using TF-IDF features along with a Logistic Regression based model:

- **Training accuracy:** 60.11 %
- **Test accuracy:** 56.53 %

4 Deep Learning Model

In this section, contrary to the Logistic Regression based model, we discuss the task of text classification through an *LSTM* (Long Short-Term Memory) based Recurrent Neural Network. We'll explain in a subsection why the modelling was done through LSTMs and not vanilla RNNs. The deep learning library, *PyTorch*, has been used to implement the architecture.

4.1 Data and Task

The data and task remain the same.

However, here, instead of fetching the data directly from the JSON files, we used the *torchtext* package. This package contains many tools and utilities to fiddle with various popular NLP datasets and respective tasks, just like we the *torchvision* package for vision problems.

The *SNLI* dataset is also present in this package and we retrieve the train, validation and test splits directly from there. Also, *torchtext* provides us the provision of tokenizing the data, lower or upper casing all the letters and even represent the words using pretrained vectorial representations like *word2vec* or *GloVe* while loading the data splits.

The tokenization is done through the *SpaCy* library and all the letters in the corpus are converted to lowercase. We describe the *word embeddings* used in the next subsection.

4.2 Network Architecture

In this approach, the task of text classification is done using an *LSTM* Model.

Instead of modelling the sequence of words in a sentence through *LSTMs*, the plain vanilla *RNNs* could've been used. However, vanilla RNNs have a big problem of *vanishing gradients* and this limits their use for longer sequences. Hence, an obvious choice was to go with LSTMs.

We now describe the architecture of the model. The neural network consists of the following 3 layers:

- **Input and Embedding Layers:** First, the premise and the hypothesis are represented using one-hot vectors and are then transformed to the pre-trained *GloVe* representations, **(glove.840B.300d)**.

  GloVe is an unsupervised learning algorithm for obtaining vector representations for words, wherein training is performed on aggregated global word-word co-occurrence statistics from a corpus, and the resulting representations showcase interesting linear substructures of the word vector space.

  The corpus used for training and obtaining these representations is the entire *Wikipedia* corpus. The 300*d* in *(glove.840B.300d)* implies that the vectorial word representations are 300-dimensional vectors. One could use various such word embeddings.

  In our model, for performing classification on the *SNLI* dataset, *GloVe* perfomed better than the other popular embedding, *word2vec* (in terms of accuracy on the test set) by a narrow margin. We discuss arriving at this model in the *Why this Architecture?* subsection.

These embeddings are pretrained for the Wikipedia dataset. Though effective, we'd like to warp the word embeddings to best suit the *SNLI* dataset.

For this, we learn these embeddings through PyTorch's *nn.Embedding()*. This method takes the cardinality of the dataset vocabulary, fabricated from the training and the validation sets, as its input and emits a 300-dimensional vectorial representation for each word.

The output dimension is a hyperparameter which is decided by the validation set provided (discussed later).

Recapitulating the above, this layer takes as input a *batch* of training examples where each word is represented by its corresponding index in the vocabulary. Each word in the input sequence is first internally converted by PyTorch into a one-hot vectorial representation, which is then further transformed to its 300-dimensional *GloVe* representation. The input layer parameters are thus initialized by the pre-trained embedding vectors of **glove.840B.300d**. This layer of the model, then learns a 300-dimensional word embedding for each word of the vocabulary encountered in the input sequence to morph the earlier *GloVe* representation to suit the dataset the best.

- **Memory Layers:**

  This layer forms the recurrent unit of the model, to deal with the input sequence of words. LSTMs were developed to deal with the vanishing gradient problem that can be encountered when training traditional RNNs.

  The output for the LSTM is the output for all the hidden nodes on the final layer. *hidden_size* - the number of LSTM blocks per layer. *input_size* - the number of input features per time-step. *num_layers* - the number of hidden layers. In total there are *hidden_size* $\times$ *num_layers* LSTM blocks.

  For each element in the input sequence, each layer computes the following function:

$$
\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\
g_t &= tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
h_t &= o_t \odot tanh(c_t)
\end{aligned}
\tag{4}
$$

  In equation 4, h_t is the hidden state at time t, c_t is the cell state at time t, x_t is the input at time t, h_{t-1} is the hidden state of the layer at time $t-1$ or the initial hidden state at time 0. i_t, f_t, g_t, o_t are the input, forget, cell, and output gates, respectively. σ is the sigmoid function, and $\odot$ is the Hadamard product.

  The specifications of this layer are as follows:

  1. **Input dimension:** 300-dimensional word embedding vector.
  2. **Output dimension:** 256-dimensional state vector.

3. **Number of hidden units:** 256.
4. **Number of LSTM cells:** 3 LSTMs concatenated layer after layer.

The output vector chosen is the sum of all the units of the last hidden layer. The next subsection also explains why the sum of the last hidden layer was chosen as the output vector.

- **Fully Connected Layers:**

  The last stage of the model consists of a set of fully connected (FC) network layers. These are the specifications:

  - **Input:** 512-dimensional input vector obtained from the current state vector added with selectively weighted previous state vectors.
  - **Output:** Probability Distribution of the class labels. Since there are 3 class labels, the distribution is a 3-dimensional vector.

  The implementation has a fully connected network of 4 layers, the fourth one being the output layer of the entire network.

  The input to this network is a concatenation of the two 256-dimensional vector outputs from the LSTM layer. These two output vectors correspond to the encoded representations of the premise and hypothesis of the dataset, which were fed independently to the LSTM.

  At first, the model was tested with only 2 fully connected layers. However, turning 2 layers to 4 increased the accuracy on the test set quite considerably without affecting the size of the model by a wide margin.

  The structure of the fully connected network is as follows:

  - **Layer 1:** Input dim - 512, Output dim - 1024, *ReLU* Activation
  - **Layer 2:** Input dim - 1024, Output dim - 1024, *ReLU* Activation
  - **Layer 3:** Input dim - 1024, Output dim - 1024, *ReLU* Activation
  - **Output Layer:** Input dim - 1024, Output dim - 3, *SoftMax* Activation

  The final 3 dimensional output gives a probability distribution over the 3 classes. The class corresponding to the max. of the 3 probabilities is chosen as the prediction of the model for the given input sentence pair.

4.3 Why this architecture?

The reason behind arriving at the above stated model is a medley of sequence modeling concepts, heuristics and empirical experiments. The model is a result obtained after several attempts to improve the performance of the model on the test set. We now describe the various attempts made to come up with this model and the pitfalls faced:

- **LSTMs vs RNNs:** Before arriving at LSTMs for modeling the incoming sequences, an Recurrent Neural Network (RNN) based architecture. The main problem associated with RNNs is the *vanishing gradient* problem. This can be seen through the evaluation of the model on the test set. The proposed model which includes LSTMs performs much better on the training set and also generalizes better on the test set: 78.60 % (LSTM) as against 73.06 % (RNN).

  The major upgrade comes in the form of the properties of LSTM gates: selectively read, write and forget.

- **Word embeddings:** The initial pre-trained word embeddings were tested with various other models before finally arriving at the 300-dimensional *GloVe* vectors. The following were tested:

  - Twitter GloVe representations: 200-dimensional.
  - Twitter GloVe representations: 300-dimensional.
  - 200-dimensional skip-gram (word2vec).

  Using the pre-trained word **(glove.840B.300d)** embeddings speeded up learning and achieved increased performance in much lesser number of iterations. This is much better than starting off from a random initialization of word embeddings and learning them from scratch, which leads to slower learning of vectorial representations and more number of epochs during training.

- **FC Layers:** Various tests on the fully connected network part of the model were also made. A shallow FC network, with only 2 layers (including the output layer) got trained in a considerably lesser time. However, the accuracy of this model on both the train and the test sets was sub-optimal and it couldn't go beyond 70 % on the test set. Gradually increasing the number of layers of the FC network resulted in a much better performance on the test set, albeit with an added cost in terms of training the model and the number of parameters. In the proposed model, we have used a 4 layered FC network which includes the output, softmax activated, layer.

- **Non-linear Activations:** $ReLU$ and $LeakyReLU$ activation functions usually perform better than their other sigmoidal counterparts. Using $ReLU$ activations both before generating the state vectors and in the FC layer led to a better performance when compared to *Tanh* and *Logistic* activations.

- **Output State Vector:** The output state vector at a timestep was chosen as the sum of all the units of the last hidden layer. This output vector gave a better performance in terms of accuracy on the test set when compared to considering:

  - the output vector of the last unit of the third LSTM layer; and
  - simple averaging of the state vector outputs of all the LSTM layers.

4.4 Loss Function

Since this is a multi-class classification task, the *Categorical Cross Entropy* loss function is a natural choice as it maximizes the probability of the true class.

4.5 Learning Algorithm and Optimizer

Since we have a Recurrent Artificial Neural Network, the learning algorithm is *Backpropagation through time* rather than the simple *Backpropagation* that we have for other models.

The *Adam* optimizer outperforms the other momentum-based optimizers and even batch stochastic gradient descent.

The betas were set to the default 0.9 and 0.999. The ϵ term added to the denominator to improve numerical stability was set to 10^{-8}.

The hyperparameter tunings are discussed in the next section.

4.6 Hyperparameters

In this section, the various hyperparameters that are present in the model or are used during the training. They are as follows:

- **Optimizer:** The Adam Optimizer was used for training the model, as was explained in the last subsection.
- **Regularizer:** A Dropout with a probability 0.2 has been used after each layer in the fully connected network. SNLI dataset also includes a validation set and hence, training was stopped when the performance on the validation set did not increase further (and started decreasing).
- **Loss Function:** To train the model, the Cross Entropy loss, with 3 classes, was minimized.
- **Number of epochs:** The max number of epochs was fixed at 90 for training, even though it was an overkill. The performance of the model on the test set saturated at around 11 epochs and the accuracy on the test set did not go beyond 78.60 %.
- **Learning Rate:** The model was trained with a starting learning rate of 0.0001. The *Adam* optimizer adjusts the learning rate after each epoch based on the values of the previous gradients.

4.7 Adding Attention

As mentioned earlier, we take the sum of all the LSTM output state vectors as the final output vector of the memory layer. There are multiple ways instead of simply taking the sum, viz., taking a simple average of the state vectors or considering only the final state vector as the output vector.

There's another way to improve the model architecture: by adding *attention* in this layer.

In a nutshell, attention can be broadly interpreted as a vector of importance weights: in order to predict or infer one element, in our case, a word in a sentence, we estimate using the attention vector how strongly it is correlated with the word under consideration.

In our case, instead of taking the simple average or the sum of the state vectors, we can take a weighted average of these LSTM output state vectors. The weights in this weighted average are nothing but the attention weights!

We denote *energy* as the importance of the i^{th} word in generating the j^{th} word. Here, we expect the attention mechanism to learn better intermediate state vectors, capturing the similarities between vectors in the latent space.

We trained the model using the attention mechanism. The accuracy achieved on the test set, however, remained in the same range. Due to this added mechanism, the number of parameters obviously increase. Since the added mechanism did not contribute much, the mechanism was not implemented in the final model.

4.8 Experiments

The model was obtained after various heuristical testings and hyperparameter tunings. Since it is not possible to list all of the results here, the notable ones are tabulated below:

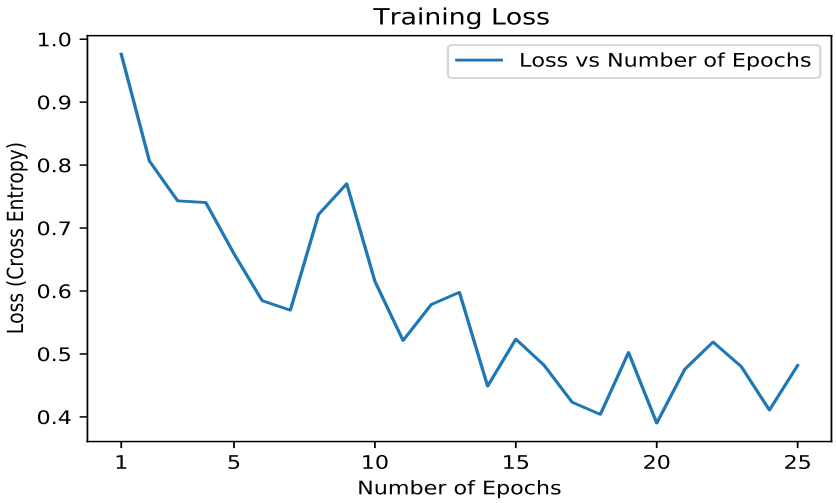
Learning Rate	**Max epochs**	**Training loss**	**Train Accuracy**	**Test Accuracy**
0.0005	90	0.659	81.05%	72.38%
0.0003	90	0.721	82.11%	74.72%
0.0004	90	0.578	82.30%	76.72%
0.0002	90	0.482	83.62%	77.14%
0.0001	90	0.391	85.96%	78.60%

In the table,

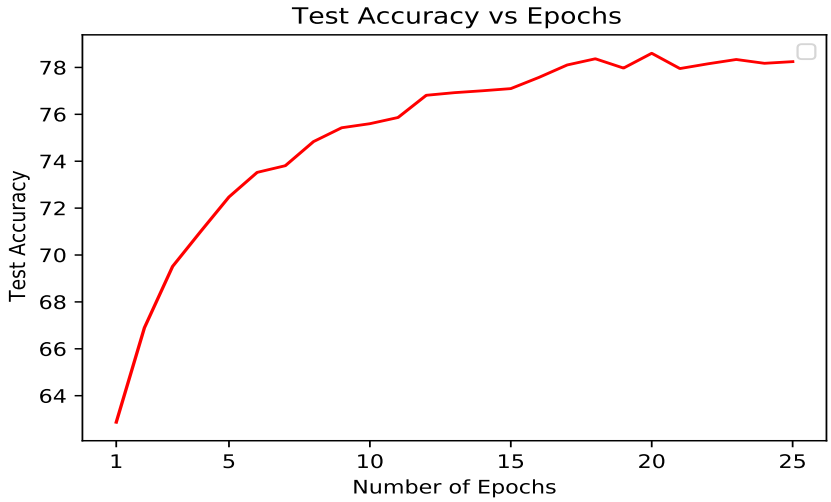
- the first result was obtained using vanilla RNNs and 200-dimensional word2vec pretrained vectors;
- the second result was achieved by using LSTMs instead of the RNNs, with the final output state vector being the last LSTM layer's output vector and 200-dimensional word2vec pretrained vectors;
- the third result was obtained by using LSTMs, with the output vector being the sum of all previous layer outputs, and also increasing the number of fully connected layers;
- the fourth result was achieved by replacing the previous 200-dimensional word2vec embeddings with 300-dimensional GloVe word embeddings (Wikipedia, 300D); and
- the fifth result combined all the positives from the above result, with some more fine hyper-parameter tunings mentioned in the previous subsection, and is the proposed model.

We now plot the results obtained from the proposed model:

1. **Loss vs Epochs:**



2. **Test Accuracy vs Epochs:**



4.9 Ablation Study

We now list down the results of the *ablation study*. Ablation study typically refers to removing some "feature" of the model and seeing how it affects performance.

1. **Removing 2 FC Layers:**

   We see the effect of removing 2 FC layers here. Hence, instead of originally having 4 FC layers, we're now left with only 2 FC layers. The accuracy of this model on both the train and the test sets was sub-optimal and it couldn't go beyond 73 % on the test set. Hence, we can draw a conclusion that the FC layers indeed play an important part in the model, and contribute towards improving the model performance.

2. **Replacing LSTMs with GRUs:**

   Gated recurrent units (GRUs) are like LSTMs with forget gate, but have much fewer parameters than typical LSTMs. This is because a GRU lacks an output gate. Replacing an LSTM with a GRU can also be a part of ablation study, as technically, we're *removing* the output gates of the LSTM cells.

   For each element in the input sequence, each GRU layer computes the following function:

$$
\begin{aligned}
r_t &= \sigma(W_{ir}x_t + b_{ir} + W_{hr}h_{t-1} + b_{hr}) \\
z_t &= \sigma(W_{iz}x_t + b_{if} + W_{hz}h_{t-1} + b_{hz}) \\
n_t &= tanh(W_{in}x_t + b_{in} + r_t \odot (W_{hn}h_{t-1} + b_{hn})) \\
h_t &= (1 - z_t) \odot n_t + z_t \odot h_{t-1}
\end{aligned}
\tag{5}
$$

   In equation 5, h_t is the hidden state at time t, x_t is the input at time t, $h_{(t-1)}$ is the hidden state of the layer at time $t-1$ or the initial hidden state at time 0. r_t, z_t, n_t are the reset, update, and new gates, respectively. σ is the sigmoid function, and $\odot$ is the Hadamard product.

   GRUs performed better than the vanilla RNNs but were not able to beat the accuracy achieved by the LSTMs on the test set. The accuracy achieved by this replacement was 75.23 % on the test set.

   This could be attributed to a study done by *Gail Weiss*, *Yoav Goldberg* and *Eran Yahav*, that the LSTM is *strictly stronger* than the GRU as it can easily perform unbounded counting, while the GRU cannot. That's why the GRU fails to learn simple languages that are learnable by the LSTM.

3. **Removing pre-trained embeddings:** Earlier in the *Network Architecture* subsection, we had described using pre-trained GloVe embeddings to represent the words in the incoming input sequences.

   Even though it is a standard approach to use pre-trained embeddings, we removed this to study its effect. The model learned the word embeddings alone from nn.Embedding() layer of PyTorch.

   It was observed that the model was still able to replicate similar results even without the pre-trained embeddings. However, for the model to achieve similar results, it took nearly $\tilde{3}5$ epochs without GloVe as against only $\tilde{2}0$ epochs with the original model.

Note that nn.Embedding() learnt 300-dimensional embeddings, as describe in the architecture. 200-dimensional word vectors obtained through word2vec or GloVe and further nn.Embeddings resulted in a sub-optimal performance.

We now summarize the results of ablation study:

Ablation	**Accuracy on the test set**	**Reduction in accuracy**
Removing FC Layers	73.06 %	5.54 %
Replacing LSTMs with GRUs	75.23 %	3.37 %
Removing pre-trained embeddings	78.50 %	0.10 %

4.10 Performance Measure

Top-1 Accuracy: The metric used to evaluate the model is Top 1 Accuracy. An example from the test set is deemed classified correctly if the max. probability of the distribution predicted by the model corresponds to the true class.

$N_{correct}$ - Number of correct predictions based on top one accuracy.

N - Total count of test data. Here we have $N = 9824$

$$accuracy = \frac{N_{correct}}{N} \tag{6}$$

The accuracy achieved using this architecture is 0.7860. As is clear, the deep model significantly improves the accuracy on the test set to 78.60 % as against the 56.53 % accuracy of the Logistic Regression model with TF-IDF features.

5 Results

- The accuracy achieved on the test set using Logistic Regression with TF-IDF features is 0.5653. The convergence occurred in around 45 iterations.
- The accuracy achieved on the test set by LSTM Network model is 0.7860. This was achieved in 20 epochs (even though the model was run for 90 epochs).